Generating adjoint expressions for Matlab

Johannes Willkomm

Institute of Scientific Computing
RWTH Aachen University

Tenth European Workshop on Automatic Differentiation

Outline

Outline

Extending our tool ADiMat with reverse mode

ADiMat implements AD *source transformation* of Matlab code.
Consider a function with signature **function** z = f(a)

- Forward mode: adimat f.m produces
  function [g_z, z] = g_f(g_a, a)
- Reverse mode: admproc f.m produces
  function [a_a, z] = a_f(a, a_z)
  via *XSL transformations*

In both cases the derivative variables can be either native doubles, which (in general) results in scalar mode or one can use one of several derivative classes, which allows vector mode.

Outline

Adjoint code rules

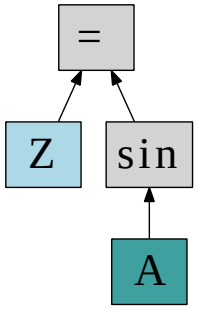
Transformation rules found in the literature are for scalar valued variables

- Consider $z = e(x)$
- Adjoint statement: $\overline{x} \mathrel{+}= \frac{\partial e}{\partial x}\overline{z}$
- What happens when $e(x) = a * x * b$ and the variables are matrices?
- What is $\frac{\partial e}{\partial x}$ in this case?
- Dimensions of $\frac{\partial e}{\partial x}$ and $\overline{z}$ will not fit in general.

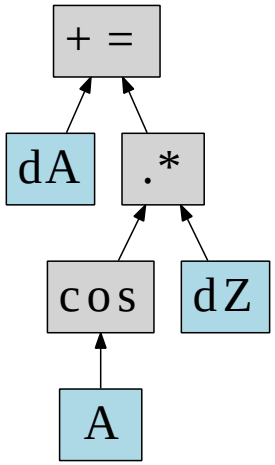
Outline

Adjoint rules: Sine

- Sine: $Z = sin(A)$



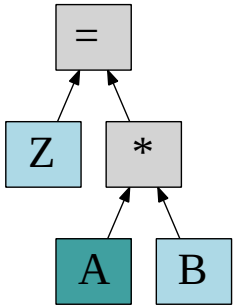
- $\overline{A} \mathrel{+}= \overline{Z} .* cos(A)$



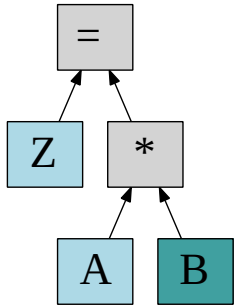
- Derivative variables have the same dimension as the variable they are associated with.
- Use upper case letters to indicate rule place holders.
- The place where $\mathrm{dZ} = \overline{Z}$ occurs is the *adjoint position*.

Adjoint rules: Multiplication

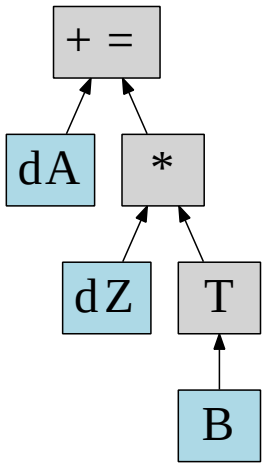
- Multiplication
  - $Z = A * B$, w.r.t. A



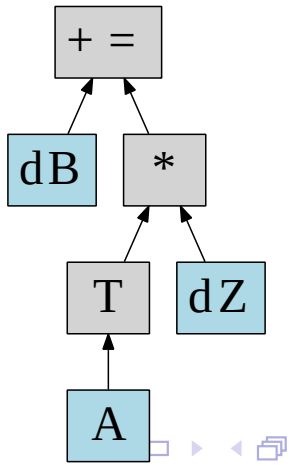
  - $Z = A * B$, w.r.t. B



  - $\overline{A} \mathrel{+}= \overline{Z} * B^T$



  - $\overline{B} \mathrel{+}= A^T * \overline{Z}$

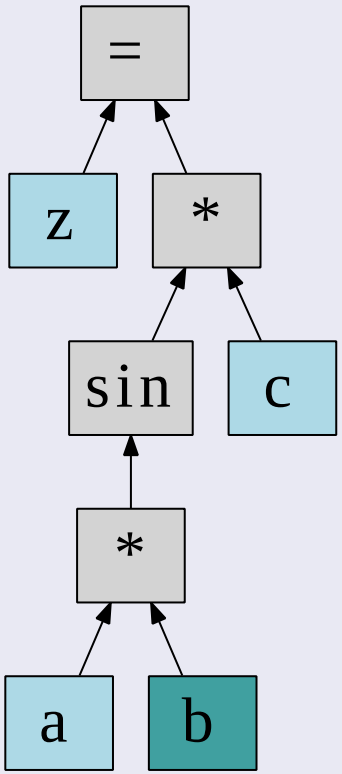


Outline

Example statement

- Consider the statement: $z = \sin(a * b) * c$

Syntax tree

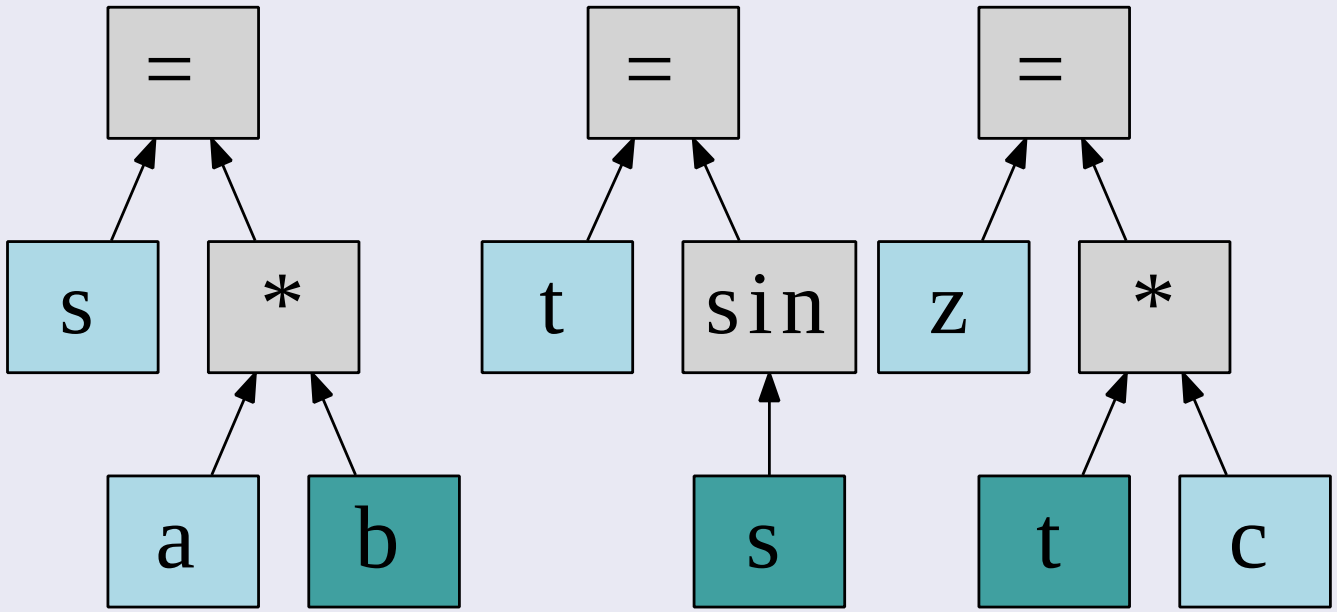


Outline

Canonicalized statement

- Canonicalization: $s = a * b$, $t = \sin(s)$, $z = t * c$

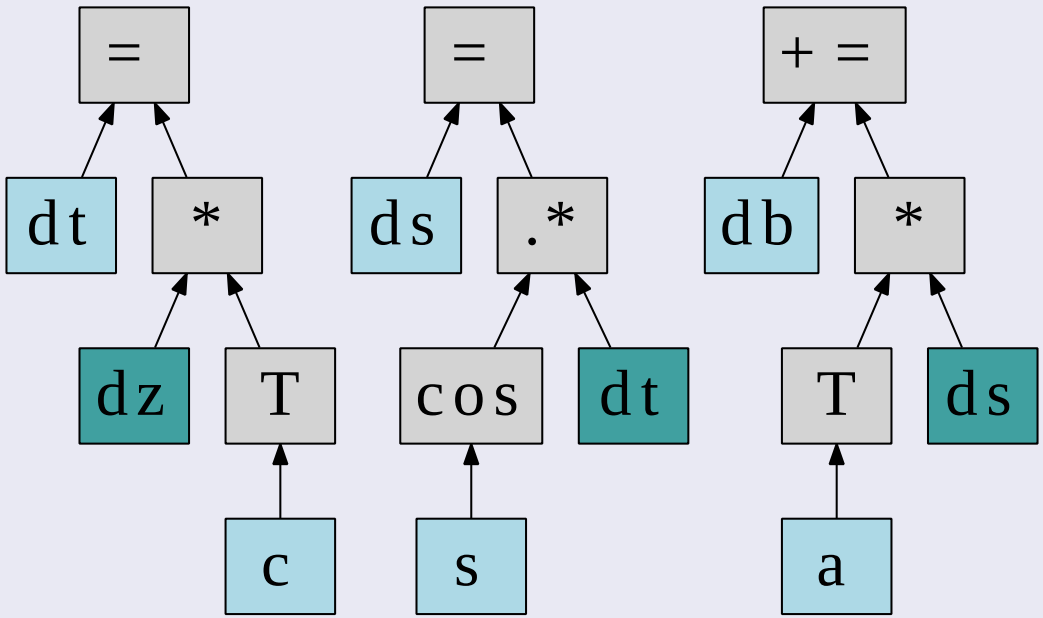
Syntax trees



Adjoint canonicalized statements

- Reverse adjoint statements:
  $\bar{t} = \bar{z} * c^T, \bar{s} = \cos(s) .* \bar{t}, \bar{b} \mathrel{+}= a^T * \bar{s}$

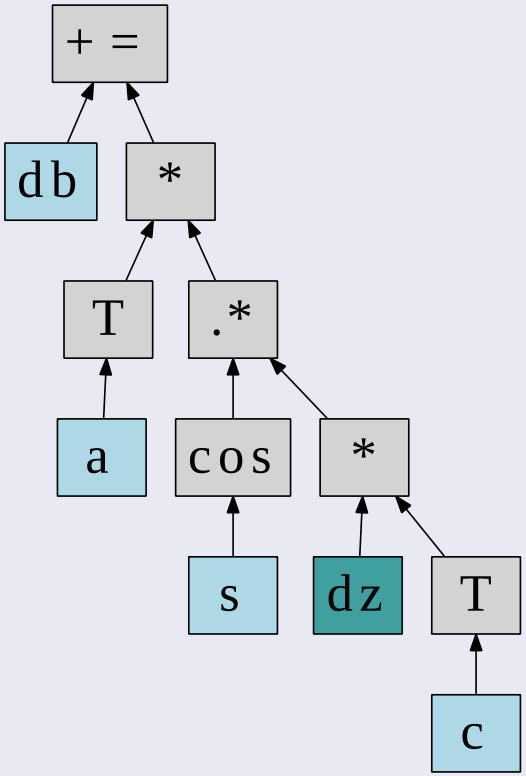
Syntax trees



Putting it back together

- Single adjoint statement: $\overline{b} \mathrel{+}= a^T * (\cos(a * b) .* (\overline{z} * c^T))$

Merged syntax tree



Outline

Observations

- The outermost $*$-operator has the adjoint of the statement LHS at the adjoint position.
- The second level operator sin has at the adjoint position the expression sub-tree that was produced by the outermost operator.
- The evalation order is inverted.

Recursive procedure

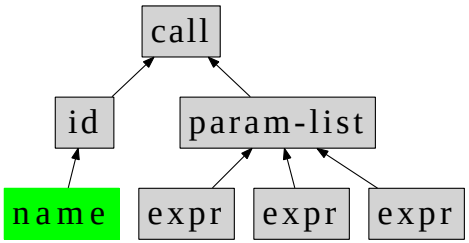
- Consider a statement $z = e$, assigning expression $e(x)$ to z. What is the adjoint expression w.r.t. x for that statement.
- Let $a_1 = \text{adjexp}(e, a_0)$ be the adjoint expression of the outermost operation in expression e, where $a_0 = \overline{z}$.
- Then, $a_2 = \text{adjexp}(\text{active-child}(e), a_1)$ is the adjoint expression of the second level operator, where active-child selects the child of the top along the path to variable x.
- Finally, a_k is the adjoint of the expression e w.r.t. variable x, where k is the depth of x in e.

Outline

Recursive template calls

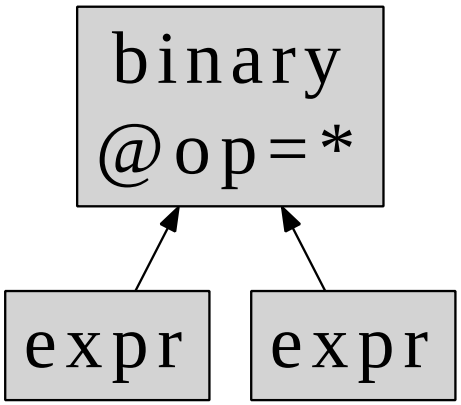
- Traverse the expression tree in top down order with apply-template
- Pass along two parameters
  - wrt – id of variable node in the expression
  - adj – adjoint constructed so far
- xsl:apply−templates to that child which has the node with id $wrt among its descendant−or−self::.
  - wrt – $wrt
  - adj – new expression according to rule for current node, inserting $adj at the adjoint position of the rule

Template For Sine



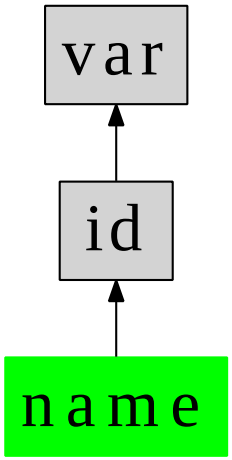

```
<xsl:template match="call[id = 'sin']" mode="diff">
  <xsl:param name="wrt"/>
  <xsl:param name="adj"/>
  <xsl:apply-templates select="*[2]/*" mode="diff">
    <xsl:with-param name="wrt" select="$wrt"/>
    <xsl:with-param name="adj">
      <binary op=".*">
        <call>
          <id>cos</id>
          <xsl:apply-templates select="*[2]"/>
        </call>
        <xsl:copy-of select="$adj"/>
      </binary>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>
```

Template For Multiplication




```
<xsl:template match="binary[@op = '*']" mode="diff">
  <xsl:param name="wrt"/>
  <xsl:param name="adj"/>
  <xsl:variable name="which"
    select="*[descendant-or-self::*[generate-id(.) = $wrt]]"/>
  <xsl:apply-templates select="$which" mode="diff">
    <xsl:with-param name="wrt" select="$wrt"/>
    <xsl:with-param name="adj">
      <xsl:choose>
        <xsl:when test="count($which/following-sibling::*)">
          <adjoint-left-multiplication>
            <xsl:copy-of select="$which"/>
            <xsl:copy-of select="$adj"/>
            <xsl:copy-of select="*[2]"/>
          </adjoint-left-multiplication>
        </xsl:when>
        <xsl:otherwise>
          <!-- ... -->
        </xsl:otherwise>
      </xsl:choose>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>
```

Template For Variable Nodes




```
<xsl:template match="var" mode="diff">
  <xsl:param name="wrt"/>
  <xsl:param name="adj"/>
  <xsl:choose>
    <xsl:when test="generate-id() = $wrt">
      <xsl:copy-of select="$adj"/>
    </xsl:when>
    <xsl:otherwise>
      <literal>0</literal>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Template For Statement

```
<xsl:template match="binary[@op = '=']" mode="adjoint">
  <!-- pop variables, ... -->
  <xsl:apply-templates mode="adjoint-assignment"
      select="*[2]/descendant-or-self::var">
    <xsl:with-param name="this" select="."/>
  </xsl:apply-templates>
  <!-- zero adjoint of variable written, ... -->
</xsl:template>

<xsl:template match="var" mode="adjoint-assignment">
  <xsl:param name="this"/>
  <xsl:param name="adj">
    <xsl:apply-templates select="ancestor::binary[@op = '='][1]/*[1]"
                         mode="adjoint-var-of-statement"/>
  </xsl:param>
  <xsl:variable name="myid" select="generate-id()"/>
  <adjoint-increment>
    <target>
      <xsl:apply-templates select="(parent::array|.)[1]"/>
    </target>
    <incr>
      <xsl:apply-templates select="$this/*[2]" mode="diff">
        <xsl:with-param name="wrt" select="generate-id(.)"/>
        <xsl:with-param name="adj" select="$adj"/>
      </xsl:apply-templates>
    </incr>
  </adjoint-increment>
</xsl:template>
```

Outline

Example: A Times B Times C

```
function z = mult3(a, b, c)
   z = a * b * c;
end
```

```
function [a_a, a_b, a_c, nr_z] = a_mult3(a, b, c, a_z)
   z = a * b * c;
   nr_z = z;
   [a_a a_b a_c] = a_zeros(a, b, c);
   a_a = a_a + a_z*(b * c).';
   a_b = a_b + a.'*a_z*c.';
   a_c = a_c + b.'*a.'*a_z;
end
```

admproc −s adjoint−reductions='no' −−nocanonicalize mult3.m

Outline

Conclusion

- Adjoints of arbitrarily nested expressions
- Ability to turn off code canonicalization
- Ability to generate code for only scalars or only matrices
- Simple implementation in XSLT

Outlook

- Represent derivative rules in one format for both forward and reverse?
- Handle cases where adjoints are given by algorithm, not an expression